National Research Council Canada
Conseil national de recherches Canada

Division of Electrical Engineering
Division de génie électrique

ERB-981

An Investigation into the Design of a Hardware Accelerator for the Harmony Real-Time Operating System

D.A. Green

March 1986

87-04605/

Copyright 1986 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Copyright 1986 par
Conseil national de recherches du Canada

Il est permis de citer de courts extraits et de reproduire des figures ou tableaux du présent rapport, à condition d'en identifier clairement la source.

Additional copies are available free of charge from:

Editorial Office, Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R6

Des exemplaires supplémentaires gratuits peuvent être obtenus à l'adresse suivante:

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R6

AN INVESTIGATION INTO THE DESIGN OF A HARDWARE ACCELERATOR FOR THE HARMONY REAL-TIME OPERATING SYSTEM

D.A. Green

ERB-981

NRCC No. 25656

MARCH 1986

TABLE OF CONTENTS

	Page
ABSTRACT/RÉSUMÉ	v
1. INTRODUCTION	1
2. PERFORMANCE OF SOFTWARE-BASED MESSAGE-PASSING IN HARMONY	3
3. HARDWARE CONFIGURATION	8
Overview	8
Interface Unit	10
Control Unit	13
Execution Unit	14
Memory Cycle Timing Estimates	19
4. SOFTWARE CONFIGURATION	23
Overview	23
Modifications to the Existing Harmony Kernel	23
Microcode Configuration	27
5. A PERFORMANCE ANALYSIS OF THE COPROCESSOR	28
A Pseudo-code Example	28
An Analysis of the Operation of the Coprocessor	28
Possible Mechanisms for Improvement	31
6. CONCLUSIONS	34
REFERENCES	36
APPENDIX A	37
APPENDIX B	41

FIGURES

1	Send-receive-reply test program	4
2	Task execution flow	7
3	Overall system configuration	9
4	Bus arbitration logic	11
5	Bus arbitration state diagram	12
6	Execution unit configuration	15
7	Memory map of register file and LRAM	18
8	Main memory read cycle timing	20
9	Main memory write cycle timing	21
10	Read-modify-write cycle	22
11	Software configuration	24
12	Proposed code for _Invoke_CP() written in 68000 assembler	25
13	Original version of _Send() written in C	26
14	Proposed version of _Send() written in 68000 assembler	26

TABLE OF CONTENTS (Cont'd)

Page

TABLES

1 A comparison of the performance of the two implementations 30
2 Coprocessor machine cycle statistics 31
3 A performance summary 33

ABSTRACT

This report describes an investigation into the development of hardware assistance for the Harmony operating system. Harmony is a multitasking, multiprocessor operating system intended for real-time control applications. Its reliance on the use of message passing between independent tasks for information exchange and task synchronization places a great deal of importance on the performance of the kernel. After analyzing that performance, a design based on newly announced components has been developed that is intended to offer a significant improvement in performance over the current implementation of the kernel. The proposed architecture is pipelined, horizontally microcoded, and employs 32-bit data paths. It uses a RAM-based register file based on the RISC architecture to support very fast context switching. The performance of the "coprocessor" has been estimated by evaluating portions of microcode that have been prepared for the proposed design. The evaluation is discussed in detail and the limitations of the proposed architecture that are identified by this analysis are discussed.

RÉSUMÉ

Le présent rapport décrit une étude portant sur le développement d'un matériel de soutien pour le système d'exploitation Harmony. Harmony est un système d'exploitation multiprocesseur multitâche destiné aux applications de commande en temps réel. Le système est fondé sur le transfert de messages entre des tâches indépendantes aux fins de l'échange d'information et de la synchronisation des tâches; la performance du noyau est donc d'une importance cruciale. Après l'analyse de cette performance, on a élaboré une structure fondée sur des composants développés tout récemment, afin d'améliorer substantiellement la performance du l'implantation actuelle du moyau. L'architecture proposée adopte une structure en pipeline à microcode horizontal, avec trajets de données 32 bits. Il exploite un fichier de registre à RAM, fondé sur l'architecture RISC, afin de permettre la commutation très rapide du contexte. On a mesuré la performance du "coprocesseur" par l'évaluation de portions de microcode préparées d'après la structure proposée. Le texte décrit en détail les travaux d'évaluation, ainsi que les limites de l'architecture proposée, identifiées grâce à cette analyse.

AN INVESTIGATION INTO THE DESIGN OF A HARDWARE ACCELERATOR FOR THE HARMONY REAL-TIME OPERATING SYSTEM

D.A. Green

INTRODUCTION

The Harmony operating system (1) was developed with two principal objectives: to provide the application programmer with a powerful abstraction to be incorporated into programs intended for real-time applications and to provide a level of performance that assures that the resulting system can be useful in such real-time applications. Harmony is a multitasking operating system that employs a message-passing mechanism to provide intertask communication and synchronization. It has been demonstrated in a sensor-based robot control application (2) and is currently being applied to other applications. This report addresses the question of performance and in particular the overhead associated with the message-passing mechanism. It documents a study undertaken to assess the performance of the message-passing process used in Harmony and outlines the high-level design of a special-purpose processor that was intended to provide a significant improvement in that performance.

Typically, computer systems can be thought of as being either "compute bound" or "I/O bound." In a message-passing multitasking system, it is reasonable to think of a system as being "message bound," particularly if message-passing consumes a significant portion of the available machine cycles. Whether or not this actually occurs is a function of the nature of the application program that is executing at any particular time and also of the fixed cost of the message-passing process. By addressing this latter issue, it is felt that the useful range of applications of a system running under Harmony can be extended.

A number of objectives were established at the outset of the project: to reduce the overhead associated with the message-passing by a factor approaching an order of magnitude, to constrain the physical size of the "coprocessor" so that it would not exceed the form-factor of a standard sized Multibus board (Multibus is a trademark of the Intel Corporation), and to minimize the impact upon the kernel of Harmony of the necessary software changes.

Harmony is currently being "ported" to other processors including the National Semiconductor 32016. It is only a question of time before it will be running on a 32-bit processor such as the Motorola M68020. Consequently, the performance of a Harmony-based system will improve in absolute terms

as these developments occur and the relative improvement as provided by a coprocessor will diminish. Therefore, a factor of ten improvement over the current software implementation of the message-passing is seen as a significant and challenging objective.

The initial implementation of Harmony runs on a system called the "Chorus" Multiprocessor (3). This is a Multibus (IEEE-796) system that uses 68000-based single-board computers (4). The coprocessor implementation was foreseen as consisting of a physical board adjacent to each processor board and cabled directly to the socket normally occupied by the processor chip. Consequently the Multibus form factor is an obvious choice.

The question of designing the coprocessor so as to minimize the impact upon the remainder of the Harmony kernel is a result of the practical limitation of available resources. A design of this nature would likely require from one and a half to two man-years to complete without the additional cost of redesigning the kernel. Therefore a precise interface mechanism has been specified that has minimal impact upon the code as it is currently configured.

This report is primarily concerned with the question of feasibility. The architecture of the proposed coprocessor consists of three major components: the interface unit (IFU) that couples the coprocessor to the 68000 processor, the execution unit (EU) that contains the data paths and ALSU and which performs the data manipulation, and the control unit (CU) which is a horizontally microcoded machine that orchestrates the operation of the entire coprocessor. The analysis concentrates on the architecture of the execution unit and makes certain assumptions about the performance of the remaining components. The decision whether or not to proceed with the design hinges upon the results obtained from the analysis of the proposed execution unit since the expected benefits of the coprocessor are almost entirely dependant upon the performance of the EU.

This report begins with the presentation of some performance data as collected from an existing Chorus machine. These data, which illustrate clearly execution times associated with message-passing, were used as a basis of comparison for the coprocessor design. Chapter 3 describes the proposed hardware configuration of the coprocessor and discusses the rationale behind the architecture. Chapter 4 outlines the proposed software configuration including the interface mechanism to the coprocessor. Chapter 5 presents some examples of "pseudo-code" which were prepared to estimate the performance of the coprocessor at the register transfer (microcycle) level and compares the performance of the two implementations. Finally, Chapter 6 discusses the results of the analysis.

2. PERFORMANCE OF SOFTWARE-BASED MESSAGE-PASSING IN HARMONY

An essential starting point was the detailed measurement of the performance of the message-passing process within the existing implementation of Harmony. The intention here was not to undertake a full analysis of the behaviour of Harmony under a wide variety of conditions but rather to establish a performance baseline in the execution of a few typical functions that can be used as a basis of comparison for the proposed design. To achieve this, a simple test program (Fig. 1) was prepared that repeatedly executed a send-receive-reply cycle. The code was designed to run on a single processor system to simplify the measurement process. By avoiding the parallelism inherent in the normal operation of a multiprocessor implementation it is possible to achieve a clearer picture of the sequential execution of the code.

Figure 1 lists the test program which is written in C. The execution of main() begins with an initialization sequence in which other tasks are created and connections to an I/O server are established. When execution resumes after the breakpoint, the "for loop" causes this task to repeatedly execute a call to _Send() that initiates the send-receive-reply cycle. A result of the execution of the _Send() function is that the task "main()" blocks and a new task is dispatched. Main() will not be ready to run again until it is sent a reply from a correspondent task, "Child()" in this case.

The task "Child()" is created by main() during its initialization. Child() consists of an infinite loop which blocks on a _Receive() function until a message is sent to it. Upon receiving a message, Child() becomes ready and when redispatched increments a counter and executes the _Reply() function. _Reply(), which is a nonblocking function, results in a message being transferred to main(), which is send blocked at this time, and the cycle repeats indefinitely.

An interprocessor interrupt precedes each message transfer. These interrupts operate at the highest maskable priority (level 6) of the 68000 processor and consequently are serviced immediately. The only exception to this is when the processor is already servicing an interprocessor interrupt and operates with all interrupts disabled for short periods. One of the results of an interprocessor interrupt is that a new task is dispatched. In this program, Child() is of higher priority than main() and so it is redispatched after the interprocessor interrupt is received as a result of the _Reply(). In this example, the messages are short: the send message is four bytes long while the reply message is six bytes.

```
extern main();
extern _Directory();
extern _Gossip();
extern Child();
extern _P0Acia0_server();
extern _Tti();
extern _Tto();
extern _Ptm_int();
extern _Serial_int();

#define CHILD           20

unsigned _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
  {
      { 1, main, 180, 7, 0 },
      { 2, _Directory, 268, 7, 0 },
      { 3, _Gossip, 226, 7, 0 },
      { CHILD, Child, 108, 6, 0 },
      { P0ACIA0, _P0Acia0_server, 412, 5, 0 },
      { TTI, _Tti, 100, 0, 0 },
      { TTO, _Tto, 126, 0, 0 },
      { 0, 0, 0, 0, 0 }
  };

void (*(_Intvec[7]))() =
  {
    0,                  /* int lvl 1 for this processor, Harmony priority 4 */
    0,                  /* int lvl 2 for this processor, Harmony priority 3 */
    0,                  /* int lvl 3 for this processor, Harmony priority 2 */
    _Ptm_int,           /* int lvl 4 for this processor, Harmony priority 1 */
    _Serial_int,        /* int lvl 5 for this processor, Harmony priority 0 */
    0, 0                /* remaining levels must be zero, will be plugged */
  };

main()
  {
    unsigned    child, id;
    struct STD_RQST     request;
    struct STD_RPLY     reply;
```

Figure 1. Send-receive-reply test program

```
    id = _Create(P0ACIA0 );
    request.MSG_SIZE = sizeof( struct STD_RQST);
reply.MSG_SIZE = sizeof( struct STD_RPLY);
    _Receive( (char *)&request, id );   /* wait for server to report */
    _Reply( (char *)&reply, id );
    _Selectinput(Open( "P0ACIA0:," READ) );
    _Selectoutput( _Open( "P0ACIA0:", WRITE ) );

    _Putstr( "parent creates child\n" );
    child = _Create( CHILD );

_Breakpoint ("after creation of child");

    _Putstr( "infinite loop underway -- bye bye\n" );
    _Flush();
    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        reply.MSG_SIZE = sizeof( reply );
        _Send( (char *)&request, (char *)&reply, child );
      }
  };

Child()
  {
    unsigned counter, requestor;
    struct STD_RQST request;
    struct STD_RPLY reply;

    counter = 0;

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        requestor = _Receive( (char *)&request, 0 );

        reply.MSG_SIZE = sizeof( reply );
        reply.RESULT = counter++;
        _Reply( (char *)&reply, requestor );
      }
  };
```

Figure 1. (Concluded)

A logic analyzer (5) was used to obtain the data. The analyzer, which included a disassembler for the 68000, was unique in that it allowed synchronous sampling of the 68000 signals as well as a small number of asynchronous channels that were clocked by an internal timebase. The display correlated the two timebases and thereby permitted the exact measurement of execution times on an instruction by instruction basis.

Figure 2 graphically presents a summary of the execution of the send-receive-reply cycle. Only the execution of Harmony kernel code is of interest here; the small portions of user code are identified but are not included in the analysis. The cycle begins with the execution of the main task (main()) which runs for a short time before calling _Send(). The primary function of _Send() is to block the execution of the currently active task (main()) and to begin the process of transferring a message to some destination task (Child()).

The actual process of transferring the message requires that the id of the originating task be written to the mailbox of the processor upon which the destination task runs, and that an interprocessor interrupt be directed to that processor. In this case, the interprocessor interrupt occurs 108.5 µs after _Send() begins to execute and of course strikes the same processor that executed the _Send(). At this point __ip_int is called to service the interrupt. For the next 58 µs (this figure includes the level 6 interrupt handling required by the 68000 processor) __ip_int runs and is responsible for transferring the contents of its mailbox into a queue (__In _id_q) of incoming task id's. Then it clears the interprocessor interrupt before calling __Td serv. The function of __Td_serv in this case is to modify the send queue of the receiving task which contains a list of tasks attempting to send messages to it and to add the receiving task to the Ready_q thereby unblocking that task. All of the above takes 175 µs and control returns to __ip_int. The next 68.5 µs are spent in __ip_int performing a dispatch which results in Child() resuming execution some 410 µs after the call to _Send() had occured.

"Child()" resumes execution in the _Receive() function where its task descriptor is modified. The message transfer occurs from the message buffer specified in the _Send() function to the buffer specified in the _Receive() function. The state of Child() is made "READY" and program execution returns to user code after 190.5 µs.

Shortly after resuming execution, Child() calls the _Reply() function. This results in the further manipulation of the task descriptor of Child() and the transfer of a message from Child() to main(), again on a buffer-to-buffer basis. At this point, the id of main() is written to the mailbox and an interprocessor interrupt is directed to the processor. All of the above takes 250.5 µs.

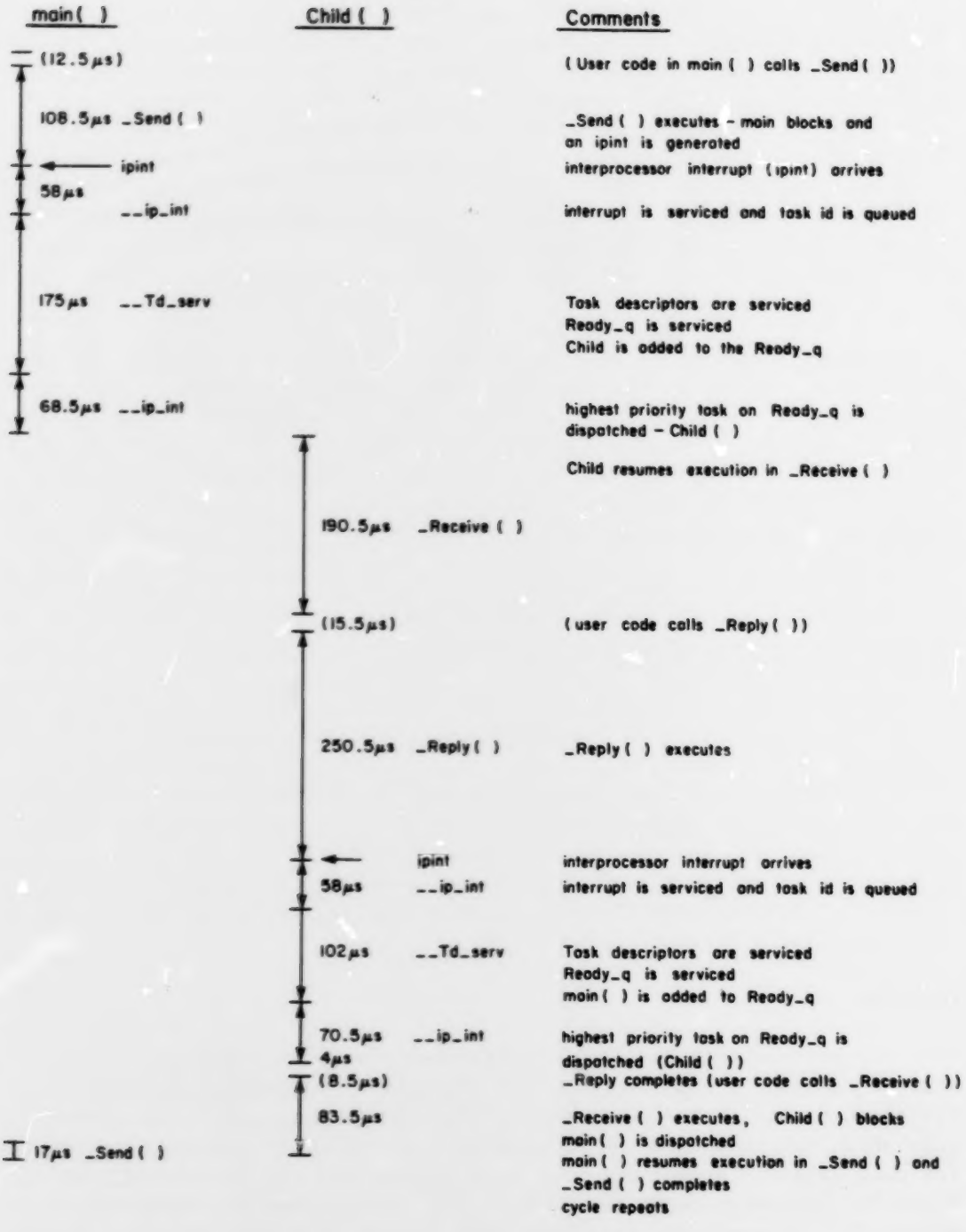
main( )
Child ( )
Comments
(12.5μs)
(User code in main ( ) calls _Send( ))
108.5μs _Send ( )
_Send ( ) executes - main blocks and an ipint is generated
ipint
interprocessor interrupt (ipint) arrives
58μs
__ip_int
interrupt is serviced and task id is queued
175μs __Td_serv
Task descriptors are serviced
Ready_q is serviced
Child is added to the Ready_q
68.5μs __ip_int
highest priority task on Ready_q is dispatched - Child ( )
Child resumes execution in _Receive ( )
190.5μs _Receive ( )
(15.5μs)
(user code calls _Reply( ))
250.5μs _Reply( )
_Reply ( ) executes
ipint
interprocessor interrupt arrives
58μs __ip_int
interrupt is serviced and task id is queued
102μs __Td_serv
Task descriptors are serviced
Ready_q is serviced
main( ) is added to Ready_q
70.5μs __ip_int
highest priority task on Ready_q is dispatched (Child ( ))
4μs
(8.5μs)
_Reply completes (user code calls _Receive ( ))
83.5μs
_Receive ( ) executes, Child ( ) blocks
main( ) is dispatched
17μs _Send( )
main( ) resumes execution in _Send ( ) and _Send ( ) completes
cycle repeats

Figure 2. Task execution flow

The arrival of this interrupt is treated in a manner similar to that described above requiring 58 µs for __ip_int and 102 µs for __Td_serv. In this case, however, the role of __Td_serv is to reactivate main() by placing it on the Ready_q. The resumption of __ip_int for 70.5 µs results in the redispatch of the highest priority task which is Child(). When reactivated, Child() only executes for 4 µs before _Reply() completes and the execution of user code continues. The user code results in a call to _Receive() which runs for 83.5 µs before blocking. The effect of _Receive() is to remove Child() from the Ready_q and to redispatch a new task, in this case main(). The resumption of main() sees the completion of the _Send() function (17 µs) and the entire cycle repeats.

The total time spent in the _Receive()-_Reply() sequence associated with Child() was 759 µs while the time spent in the _Send sequence associated with main() was 427 µs. The total time spent in the send-receive-reply cycle was 1186 µs. In general, this is a reasonable approximation to the actual overhead associated with message-passing. Variations on this figure can occur due to such factors as message length and the priorities of the tasks involved which affect the length of the search through the ready queues.

As a design objective then, the execution of the above code on the proposed coprocessor should approach 120 µs.

3. HARDWARE CONFIGURATION

This chapter outlines the organization of the hardware of the coprocessor. The principal objective of this design is that the combination of the main processor and the coprocessor should result in a throughput improvement approaching an order of magnitude in the execution of the message-passing in Harmony. A further objective was to limit the implementation of the hardware to a single circuit board.

Overview

The architecture has been partitioned into three components (Fig. 3): the execution unit that contains the data paths, register file, ALSU (arithmetic, logic, and shift unit), and related components, the interface unit which interconnects the coprocessor and the 68000 and which is required for data transfer operations with main memory, and the control unit which is a conventional microcoded achitecture that coordinates the behaviour of the entire coprocessor. The principal objective of this study was to estimate as accurately as possible the performance of the proposed design and consequently most of the effort has been concerned with the design of the

execution unit. It will be discussed in detail. The remaining components are less critical at this time and therefore will only be described in a general manner.

In concept the coprocessor is merely an extension of the main processor. The physical interface with the 68000 processor involves removing the 68000 from its socket and replacing it with a 68000 and additional hardware constructed on an adjacent circuit board. The two boards will be cabled

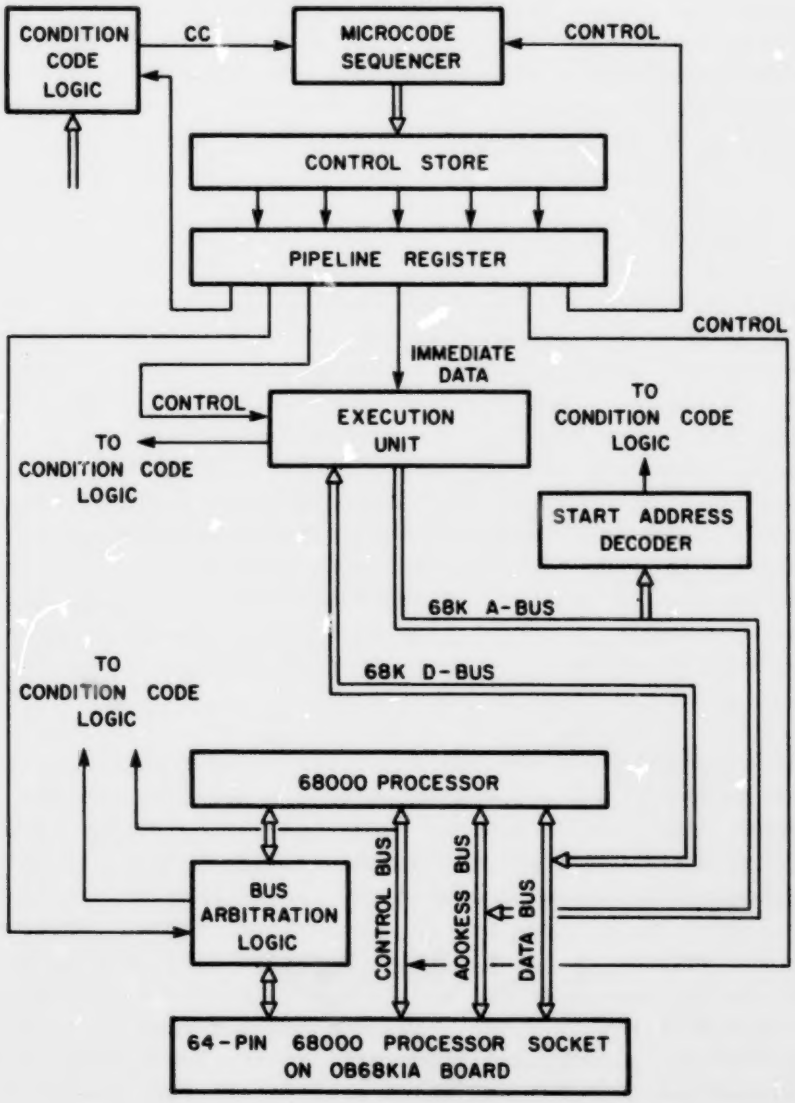

Figure 3. Overall system configuration

together via the 64-pin socket. The coprocessor will not be a bus device on the IEEE-796 bus but will derive its power from the bus.

Interface Unit

During normal operation of the main processor, the coprocessor will be inactive and transparent to the behaviour of the 68000. The opposite is true in the case where the coprocessor is operating, i.e., the 68000 will be idle as the coprocessor executes. The interface mechanism is responsible for the smooth transition between these two cases. In addition, when the coprocessor is running the interface unit will emulate the behaviour of the control signals associated with the 68000. Physically the IFU will contain the data drivers and receivers, the bus arbitration logic, the start address decoder, and access to the condition code logic. Much of the functionality of the IFU will be provided by microcode.

Transfer of Control

Transfer of control to the coprocessor will make use of the normal bus arbitration cycle of the 68000 (6). The 68000 transfers control by writing to a predetermined address that is recognized by the start address decoder of the IFU. The coprocessor then proceeds to initiate a bus request cycle by asserting bus request (BR*). The main processor asserts its bus grant (BG*) signal to indicate that the transfer has been internally synchronized and may continue. After it is recognized that the previous master has released the bus the interface unit then asserts bus grant acknowledge (BGACK*) which completes the transfer cycle. At this point, the 68000 becomes dormant and tri-states its address and data buses as well as many of its control signals, thereby allowing the coprocessor to effectively replace the main processor. On completion of its operation, the coprocessor returns control to the 68000 by negating BGACK*.

In the simplest case, the bus arbitration cycle can be directly implemented in microcode. However, the processor boards used in the system contain dynamic RAM's which must be refreshed periodically. The designers of the OB68K1A board use a DP8409 DRAM controller chip that performs this role. The logic identifies processor I/O cycles that do not access the local DRAM and in these cases the DP8409 performs hidden refresh cycles at no cost to the system. A 555 timer is used that times out at 15 µs if no such hidden cycles have occurred and initiates a forced refresh cycle which includes a bus arbitration cycle. The coprocessor must be capable of the same functionality. Consequently bus arbitration logic must be derived that will be implemented on a PAL and which will direct the flow of the control signals (Fig. 4) that interconnect the 68000 processor, the OB68K1A board, and the coprocessor's pipeline register and condition code logic.

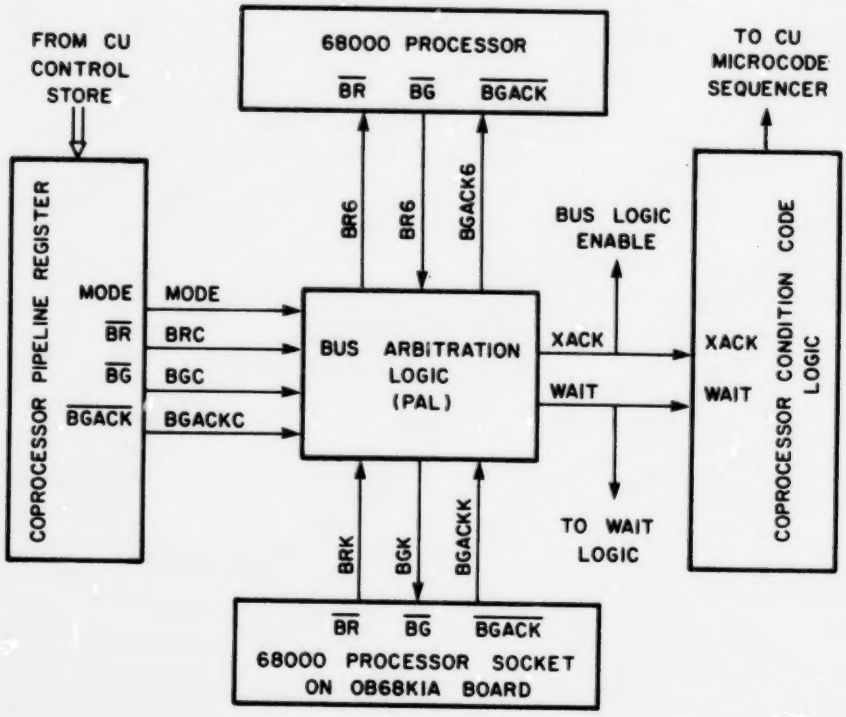

Figure 4. Bus arbitration logic

The state diagram is illustrated in Figure 5. Note that Figs. 4 and 5 are only representations of a design for the bus arbitration logic. A complete design must also include the 68000 processor's AS*, DTACK*, and BGACK* signals which are needed to determine when a previous master has terminated its bus cycle and released the bus. Nine states exist. The normal operation of the 68000 processor occurs with the arbitration logic in state 1. (Mode from the coprocessor defaults to Mode0.) A request for a forced refresh cycle from the DRAM controller causes transitions through states 2 and 3. The arbitration logic will exist in state 4 during the actual refresh cycle and returns to state 1 when the refresh cycle has completed. States 5 and 6 occur when the coprocessor is requesting control in response to a start instruction from the 68000 (mode 1 is asserted by the coprocessor here) and in state 7 the coprocessor has gained control. If the coprocessor completes its operation it returns control to the 68000 by negating BGACKC* which forces a transition back to state 1. If a DRAM refresh cycle is requested while in state 6, the state machine switches through states 8 and 9 where the refresh controller has access to DRAM. In state 9, the coprocessor is inhibited from performing memory cycles to main memory. This would likely be implemented with a hardware disable that forces coprocessor wait states. The coprocessor will only be able to drive

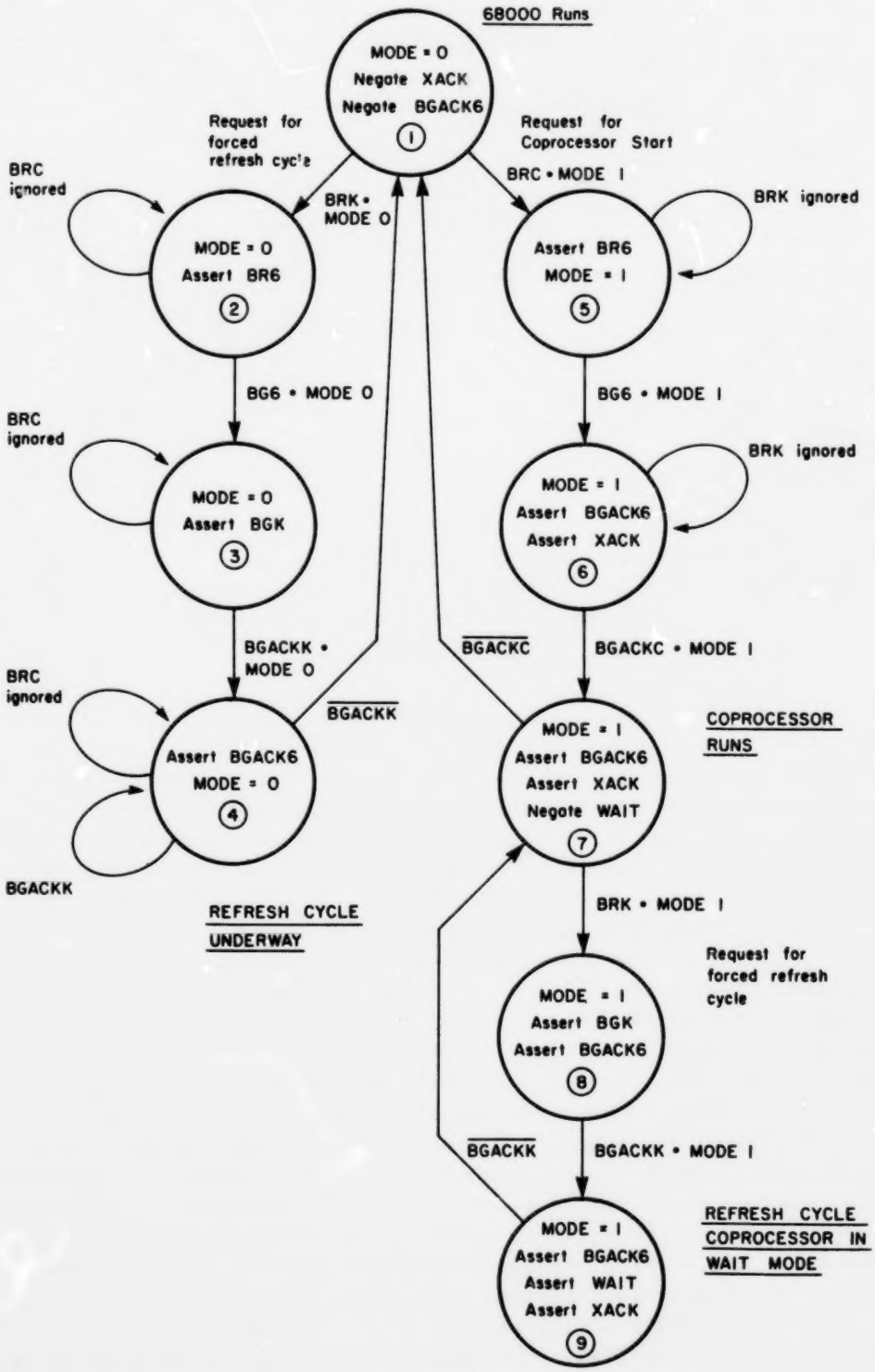

Figure 5. Bus arbitration state diagram

the 68000's address, data, and control buses when XACK (transfer acknowledge) is asserted by the arbitration logic. The details of this hardware have not been determined. Note that during the transition phase of states 5 and 6 refresh requests are ignored while during the transition through states 1 and 2 transfer requests from the coprocessor are ignored. This is not likely to cause difficulties since these states are only expected to occupy a few machine cycles.

Implicit in this discussion is the requirement for either a periodic polling loop to check for the assertion of the bus request or a micro-interrupt mechanism to perform a similar role. As described above, a synchronization mechanism has to exist to guarantee that the transfer of bus control is performed reliably; i.e., that the coprocessor has in fact completed its current bus cycle before releasing control. These details have yet to be addressed.

Memory Cycles

The coprocessor must be capable of performing memory cycles which look to the existing hardware as though they were generated by the 68000 itself. The execution unit will generate and accept the data that are being transferred but the interface unit is required to produce the correct sequence of control signals. These include the generation of address and data strobes (AS*, UDS*, LDS*), read/write signals (R/W*), and the function code signals (FC0-FC2). During any memory cycle, the interface unit must monitor the data transfer acknowledge (DTACK*) signal as well.

Interrupts

The only system interrupts that the coprocessor will be required to handle are the interprocessor interrupts. These will be identified only by occasional polling at particular points in microcode. These points will be choosen such that the contents of the internal registers of the coprocessor will not be important. Therefore the coprocessor context will not have to be saved. It was described earlier that bus requests associated with DRAM refresh cycles may be invoked by interrupts at the microcode level. These are the only interrupt sources that have been anticipated at this time.

Control Unit

This component (Fig. 3), which consists of the microcode sequencer, control store, pipeline register, and condition code register, has received the least amount of attention. Only two factors are of concern at this time: the cycle time of the controller must be 100 ns and the on-board stack of the sequencer must be at least 16 levels deep. This latter

requirement is related to the fact that the register architecture to be described below is intended to allow up to 16 levels of nesting and therefore the sequencer must also achieve this depth. The coprocessor by necessity will be horizontally microcoded and be of a single level pipeline configuration. This allows instruction fetching to completely overlap instruction execution. At this point no attempt has been made to estimate the microcode width. The width of the condition code vector still has to be determined as well.

An operational detail that has not been resolved concerns the development mechanism for the controller. In its final form, the coprocessor will be completely transparent to the user of the Chorus machine and so the microcode will likely be burned into PROMS. This method is unsatisfactory during the development phase and so either a writeable control store (WCS) or a ROM simulator will be required. The WCS requires that additional hardware be developed so that microcode can be downloaded from a host computer.

Execution Unit

This component ultimately determines the performance of the coprocessor. A number of design decisions have been taken in an effort to maximize that performance:

A full 32-bit architecture is planned which includes the data paths, the ALSU, and the register file.

As described earlier, the execution unit is pipelined with the control unit which approximately doubles the performance when compared to the more conventional von Neumann architecture.

A register file configuration based on the Reduced Instruction Set Computer (RISC) architecture has been incorporated here to allow very efficient context switching to occur.

The data paths of the execution unit are bus structured rather than multiplexer-based to reduce propagation delays.

Many logic components will be implemented with PAL's not only to improve performance but also to reduce the parts count.

The ALSU

Figure 6 describes the proposed architecture of the execution unit. The machine employs a full 32-bit data path. The focus of the design is the

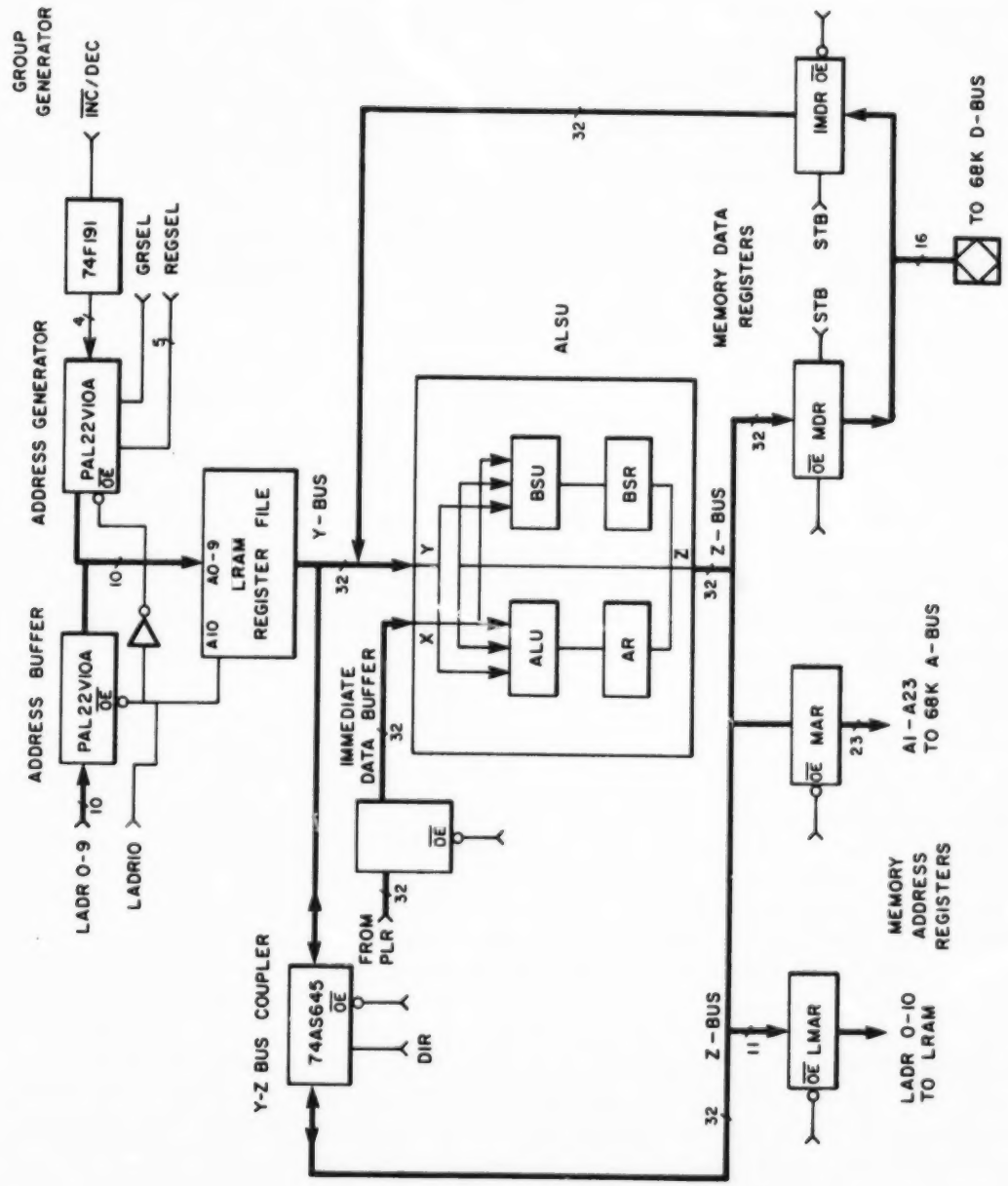

Figure 6. Execution unit configuration

32-bit ALSU which is based on the proposed ADSP-1201 from Analog Devices (7). Another device that has been considered for the ALSU is the Am29332 from AMD (8). Both devices, which each contain two input ports and a single ouput port as well as a barrel shifter, are designed to operate in systems that cycle in under 100 ns. The ADSP-1201 is a cascadable 16-bit CMOS part while the Am29332 is a non-expandable 32-bit part. While the Am29332 supports variable length field operations very efficiently, it suffers from several limitations that render it less appropriate for this design. The Am29332 contains no internal data feedback path, no input latches, and no accumulate register at its output. Consequently, the use of this part would require additional design effort to incorporate these components. While cascading two ADSP-1201's to form a 32-bit ALSU will see a slight reduction in performance, the device is still expected to achieve a 100 ns cycle time.

The ADSP-1201 ALSU consists of an arithmetic logic unit (ALU) in parallel with a barrel shift unit (BSU) which share the same data sources. The output of the BSU feeds a barrel shift register (BSR) whose contents can be directed either back to the input of either the BSU or ALU or to the output of the ALSU chip itself. An on-board register file consisting of two sets of 32-bit registers will not likely be used in the design primarily because a larger register file can be easily incorporated into the proposed memory design (see description of register file below).

Data Paths

Two buses will exist on the execution unit. The Y-bus feeds the Y input to the ALSU and the Z-bus carries output data from the ALSU. Sources on the Y-bus include: the LRAM/register file, the input memory data register (IMDR) and the Z-bus via a bidirectional driver "Y-Z bus coupler." The X input to the ALSU is sourced only by immediate data supplied directly from the pipeline register contained within the control unit. (Because of this link between the control unit and the data paths this architecture is referred to as a "modified Harvard" architecture.) The Z-bus drives the memory data register (MDR) that feeds data to the 68000 data bus, the memory address register (MAR) that feeds addresses to the 68000 address bus, and the local memory address register (LMAR) that supplies addresses to the coprocessor's local RAM. The use of the Y-Z bus coupler to link the Y-bus to the Z-bus provides a short-cut between the register file and any of the output registers (LMAR, MAR, MDR) which saves machine cycles during these data transfer operations.

In order to achieve the 100 ns cycle time it is necessary to minimize the propagation delays between memory and the ALSU. Expanding the data paths to provide a wider selection at the X input of the ALSU would introduce additional delays that would reduce the performance of the coprocessor.

Register File and Local RAM

The expense normally associated with context switching has led to the development here of a memory-based register file. This architecture which is patterned after that used in RISC machines (9) results in the execution of a context switch in one machine cycle (100 ns). This, and the fact that parameters are passed automatically because of the use of overlapping register sets, is one of the significant features of the design.

Physically the register file occupies nearly sequential addresses in a local high speed static RAM. The addressing logic generates the appearance of a register file. The address space of the RAM which is organized as 2K × 32 is divided in half with the lower portion allocated as free RAM and the upper portion forming the register file. The registers are divided into two categories: global registers, which are globally accessible from any context, and normal registers, which are associated with a particular coprocessor context and which are automatically allocated and deallocated. The decision to use the RAM-based global registers instead of the register file available on the ADSP-1201 was based on the fact that the on-board register file contains only 16 registers and that the RAM is far from being overcommitted. The design allows for 16 groups of normal registers in which each group contains 32, 32-bit registers (R0 - R31). As indicated in Fig. 7, there is an overlap of four registers between each register group (this number is subject to revision). When a new context is called for, the addressing logic automatically equates the new R0 with the previous R28, the new R1 with the old R29, etc. Therefore, transferring parameters from one context to the next requires that the caller deposit the parameters into its registers R28 - R31 in order that the callee can receive those same parameters in its registers R0 - R3. Returning arguments from a callee to a caller follows the opposite course in which the values to be returned are deposited in the lowest four registers in the register group and are recovered by the caller in its highest four registers after its context is restored. Note that with 16 register groups it is possible to nest context switches to 16 levels and that this is consistent with the design of the microcode sequencer. It is expected that this depth will be sufficient. Global registers are identified as GR0 - GR31. Note that with this configuration, some 540 long words (2160 bytes) of the RAM will not be used.

The implementation of this register file is illustrated in Fig. 6. The RAM is based on 35 ns 2K × 8 RAM's (TMM2019-35 or CY7C128-35C) which multiplex their input/output data lines. The heart of the logic will be an Am22V10 PAL which will perform the address generation/translation function. A 4-bit counter using a 74F191 will act as the "Group Generator" that essentially provides a pointer to the current base of the register file. Switching to the next or previous context physically requires merely incrementing

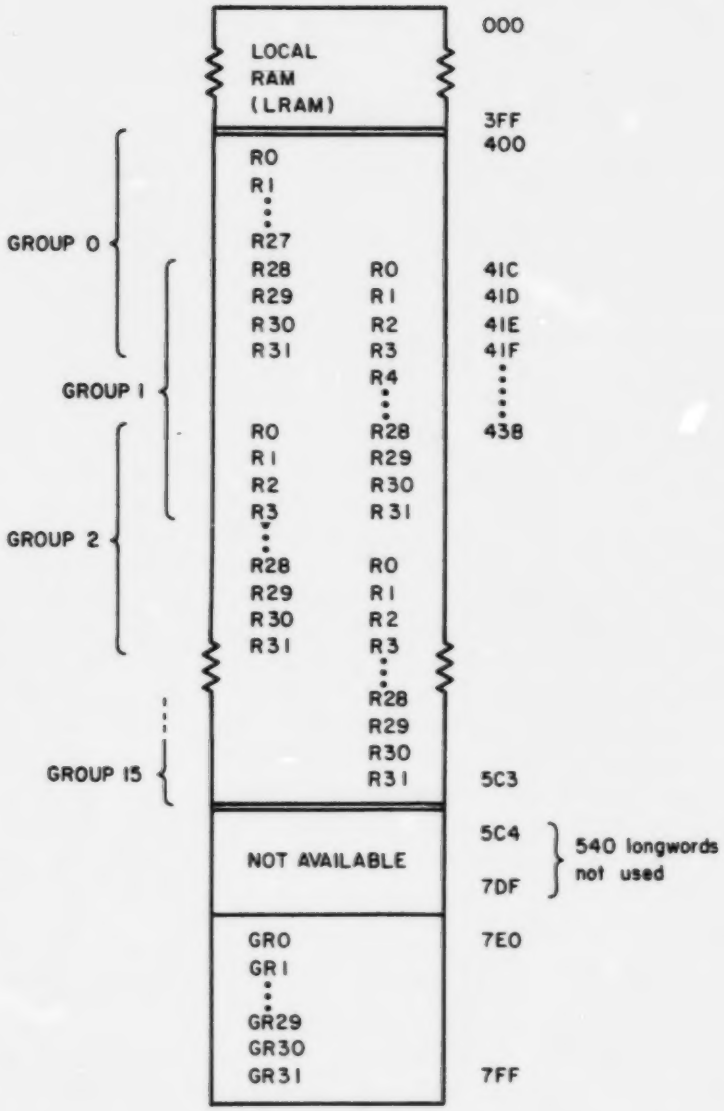

Figure 7. Memory map of register file and LRAM

or decrementing this counter. The 5-bit register select code, (REGSEL) input to the address translator PAL is combined there with the 4-bit group select code to produce the appropriate 10-bit address for presentation to the RAM (the MSB of the RAM address, i.e., A10, is used to partition the RAM between the register file and the LRAM area). Global registers will be selected by asserting GRSEL in combination with the register to be selected (REGSEL).

Data and Address Interface to the 68000

Data transfers between the execution unit and main memory involve the use of three registers. The two output registers employ tri-state outputs. The MAR accepts the lowest 24 bits from the Z-bus and presents all but the LSB to the address bus of the 68000. The MDR is used for write cycles to main memory on the OB68K1A and across the bus as well. Because the 68000 only provides a 16-bit data path (the IEEE-796 bus is also limited to 16-bit transfers) the MDR is forced to multiplex all long word (32-bit) transfers. The MDR is loaded from the 32-bit Z-bus but each of the two 16-bit words requires a separate memory cyle. The IMDR is used in the execution of memory read cycles from main memory or across the IEEE-796 bus. Again, the transfer of a long word requires two complete memory cycles at a rate determined by the memory design. The data received by the IMDR appear on the Y-bus for transfer to the LRAM, register file or the ALSU.

Memory Cycle Timing Estimates

Accessing local registers or LRAM (i.e., the coprocessor's private memory) will typically require two cycles for a long word read operation (this includes one cycle for latching the data when this is necessary) and one cycle for a long word write operation.

Detailed timing estimates have been prepared for coprocessor memory cycles that access main memory. The information described here is mainly a function of the interface unit. Figures 8, 9, and 10 illustrate read, write, and read-modify-write cycles, respectively. Read and write memory cycles are word (16-bit) operations while those for read-modify-write cycles are byte operations only.

The basic machine cycle is 100 ns. It is expected that five machine cycles will be required for each memory read cycle, assuming that a wait state will be necessary because of the 120 ns RAM's used on the OB68K1A. Additional wait states will be required for off-board (Multibus) memory cycles. During cycle one, the MAR is enabled on the falling edge of CPA*, the single phase 10 MHz system clock (see discussion of timimg details below). R/W* is set HI, and the function code FC0-FC2 is asserted. During cycle 2, the address strobe, upper data strobe, and lower data strobe (AS*, UDS*, and LDS*) are all asserted and a wait state is inserted in cycle 3. DTACK* arrives during cycle 4 indicating that memory data are valid and, in cycle 5, IMDR is strobed to latch the incoming data and AS*, UDS*, and X LDS* are all negated, completing the cycle in 500 ns.

The write cycle is somewhat similar. Cycle 1 of a write cycle requires that the MAR be enabled from the falling edge of CPA* and that the function

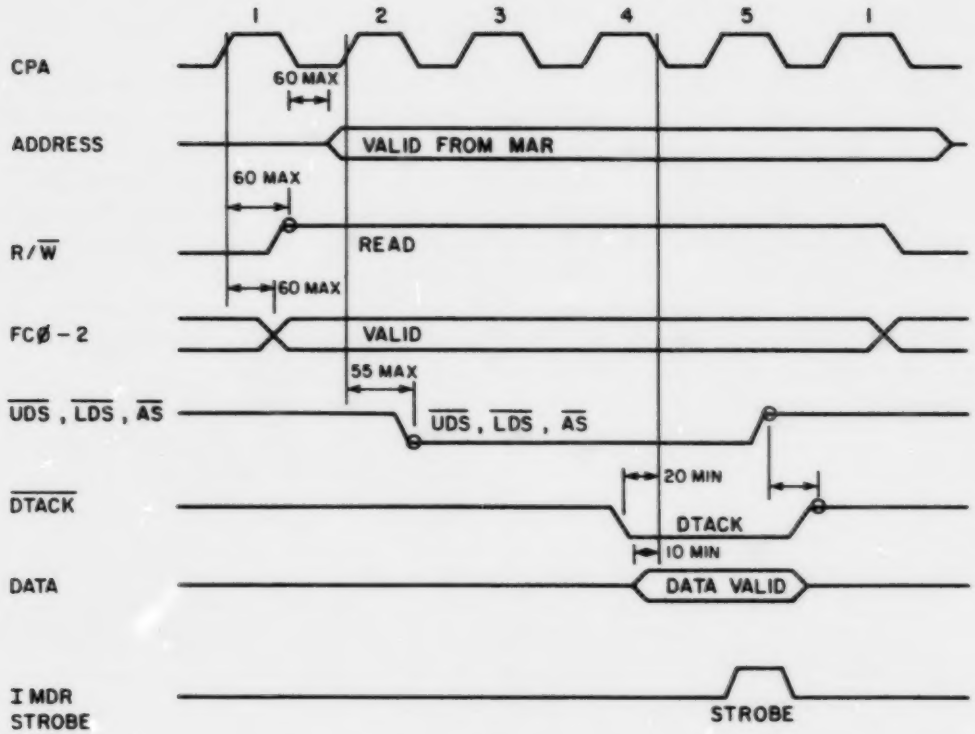

Figure 8. Main memory read cycle timing

Cycle	IFU Operation (10 MHz 68000)
1	enable MAR, assert R/$\overline{W}$ to R, assert FC0-2
2	assert $\overline{UDS}$, $\overline{LDS}$, $\overline{AS}$
3	wait state
4	receive $\overline{DTACK}$
5	strobe data into IMDR, negate $\overline{UDS}$, $\overline{LDS}$, $\overline{AS}$

code be valid. In cycle 2, AS* is asserted R/W* is set LO. Also, the MDR is enabled relative to the falling edge of CPA*. UDS* and LDS* are asserted during cycle 3 and cycle 4 sees the arrival of DTACK*. In cycle 5, the strobes are all negated (AS*, UDS*, and LDS*) to complete this cycle in 500 ns as well.

The read-modify-write cycle differs from those described above because it locks the 68000 bus for its entire duration and because it operates only on a single byte. In the implementation of the Harmony kernel, this cycle is used only in one situation; for the implementation of the TAS (test and set) instruction immediately before a mailbox is to be written to. In this

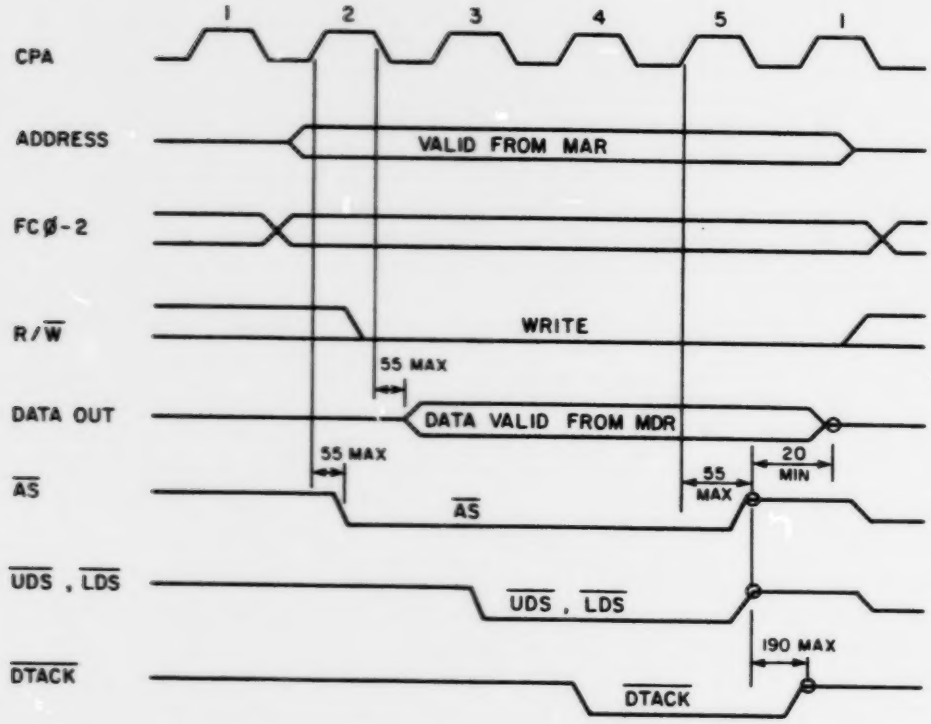

Figure 9. Main memory write cycle timing

Cycle	IFU Operation
1	assert FC0-2, enable MAR (address out)
2	assert $\overline{AS}$, R/$\overline{W}$ to $\overline{W}$, enable MDR (data out)
3	assert $\overline{UDS}$, $\overline{LDS}$ await $\overline{DTACK}$
4	receive $\overline{DTACK}$
5	negate $\overline{AS}$, $\overline{UDS}$, $\overline{LDS}$
	(MDR, MAR, R/$\overline{W}$ are automatically adjusted in next cycle)

case, during the modify phase, the MSB of the mailbox location has to be tested, an operation requiring an estimated four machine cycles (400 ns). The 500 ns read phase is similar to that described earlier except that AS* is held LO for the duration of the entire cycle. The same is true of the write phase which is also expected to last for about 500 ns. The entire cycle will take about 1.4 µs to complete.

It is clear from the above that accessing main memory is far more expensive than accessing local memory, by a factor between five and ten to one. The signifigance of this will be explained later in this report.

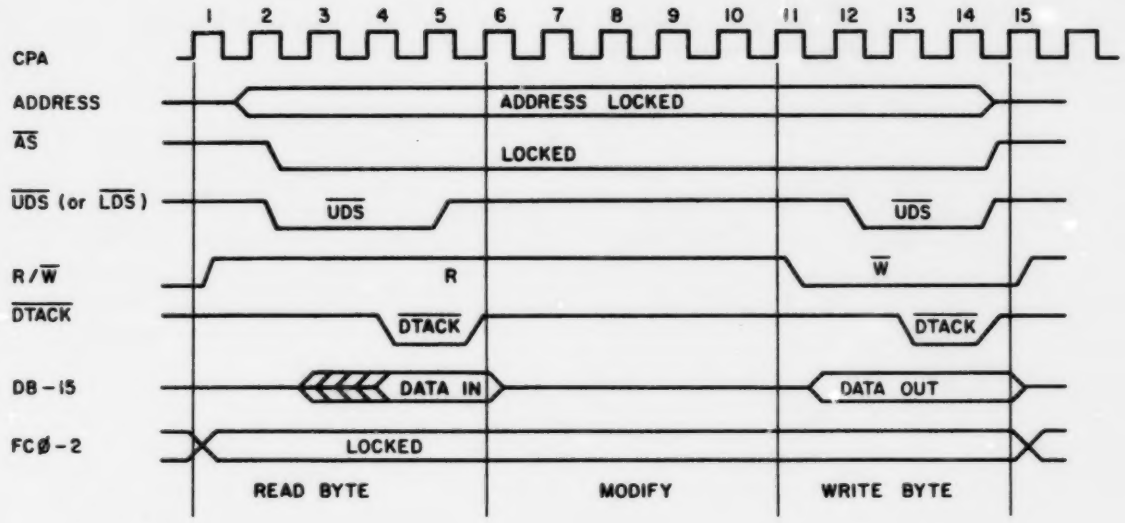

Figure 10. Read-modify-write cycle. Notes: 1. modify phase requires from three to five states; 2. write phase requires only four states since address, FC0-2 and $\overline{AS}$ are locked

Additional details related to timing estimates will be discussed here. Three factors must be accounted for when designing the strobe logic for the MAR, MDR, and the LMAR. First, because of the output delay from the accumulate register (AR) within the ALSU, the data on the Z-bus will not be valid on the rising edge of CPA* (i.e., at the beginning of any machine cycle). In fact the preliminary specifications for the ADSP-1201 do not specify the delay at all. It is assumed here that clocking the data from the Z-bus on the falling edge of CPA* (50 ns into a machine cycle) will satisfy the delay requirement. This affects the loading of the MAR, MDR, and the LMAR. If it turns out that the delay from the AR exceeds 50 ns, then these registers will have to be clocked by an intermediate clock phase that will have to be generated specifically for this purpose. Second, during a write cycle, R/W* must be setup at least 20 ns before the data are enabled. Since R/W* is derived from the pipeline register and is valid at about 15 ns into the machine cycle, it is necessary to delay the enabling of the MDR. This is conveniently achieved by clocking and enabling the MDR as described above. Third, during either type of cycle it is necessary to present a valid address within 60 ns of the falling edge of CPA*. Again this is achieved automatically by clocking the MAR as describrd above.

In presenting immediate data to the input of the ALSU it is necessary that they be valid on the rising edge of CPA*. Therefore these data have to be pipelined with the preceding machine cycle. This can be referred to as the "Pre-immediate" requirement.

4. SOFTWARE CONFIGURATION

Underlying the design of the hardware that has been described in this report is the recognition that a consistent and well-defined framework has to exist in order that the coprocessor microcode can run effectively. The situation is complicated by the fact that the flow of program execution is not always obvious given the dynamic nature of multitasking. The ability to perform fast context switches is crucial to this design and is to a large extent responsible for the simplicity of the microcode configuration. This section outlines the organization of the microcode and the companion software that is resident on the 68000.

Overview

The coprocessor is designed to perform a specific and limited role in the execution of programs running under Harmony. These functions include the response to incoming interprocessor interrupts with the level six global interrupt handler (_flgi6h), and the execution of all operations associated with the three principal message-passing primitives, i.e., _Send(), _Receive(), and _Reply(). From the point of view of the 68000, these correspond to the four entry points to the coprocessor microcode. In addition, when the coprocessor is active, it will be capable of responding to interprocessor interrupts itself without having to transfer control back to the 68000. This is expected to result in a much better response time in certain cases. The explicit initialization of the coprocessor will require a separate microroutine which will be invoked during system startup.

Modifications to the Existing Harmony Kernel

Most of the changes to the kernel of Harmony will be restricted to the four primitives mentioned above. However, two assembly language routines will be added to the kernel. The first, _Invoke_CP(), will provide a single interface between the 68000 and the coprocessor and the second, _Init_CP, will force an initialization of the coprocessor. The transfer mechanism requires the use of a communications table, _Comtab, which resides in main memory, to store parameters that are to be passed to the coprocessor. In addition, the _Comtab will be used to return values to the main processor and to identify error conditions that may arise in the operation of the coprocessor. The _Comtab has to be added along with fixed pointers to allow the coprocessor to locate pointers to _Active and the _Comtab.

Figure 11 presents the proposed configuration. Each of the four primitives that were originally executed on the 68000 will be rewritten in order to shift the processing to the coprocessor. The interface between these four routines and the microcode that runs on the coprocessor will consist of a new routine called _Invoke_CP() which is listed in Fig. 12. The purpose

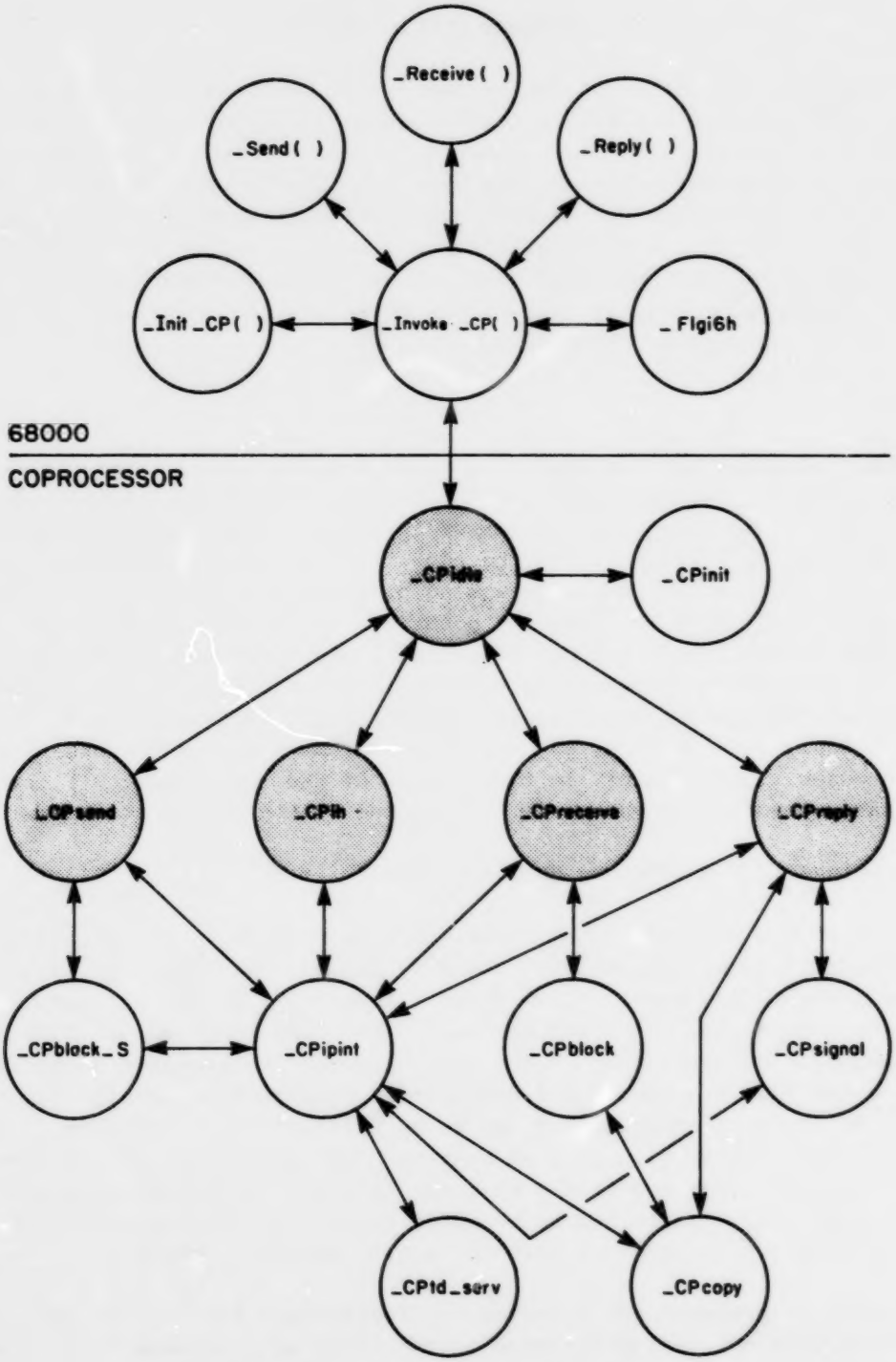

Figure 11. Software configuration

```
                    .text
                    .globl  _Invoke_CP
*
*
        _Invoke_CP:
                movea.l _Active, a0
                mfsr    -(sp)                   * save status
                movem.l d0-d7/a0-a6, -(sp)      * save registers
                move.l  sp, STACK(a0)           * save stack pointer
                move.w  #3210, _Cop             * write to coprocessor
                nop                             * transfer to cop
                                                * 68000 stops here
                nop
                movea.l _Active, a0             * _Active is restored by cop
                movea.l STACK(a0), sp           * restore sp
                movem.l (sp)+, d0-d7/a0-a6      * restore registers
                rte                             * restore status, return

                        .globl  _Active
                        .globl  _Cop
```

Figure 12. Proposed code for _Invoke_CP() written in 68000 assembler

of _Invoke_CP() is to save the entire state of the 68000 and to transfer control to the coprocessor. This involves saving all registers including the status register on the current stack and then saving the stack pointer on the task descriptor of the currently active task. This is followed by a write to an address that is decoded by the coprocessor hardware and which results in the transfer of control. The 68000 executes a NOP to allow time for the transfer to occur. All of this will take about 21 µs. When the coprocessor completes its operation it transfers control back to the 68000 which restores its state by fetching its stack pointer from the currently active task descriptor and by restoring its registers from the stack. Note that in general the task that is dispatched after a call to the coprocessor is not the same task that initiated the call. This action is expected to take just under 20 µs.

To further illustrate the nature of the changes to the kernel, consider the proposed changes to _Send(). Figure 13 lists the original source code of _Send() which is written in C while Fig. 14 presents the proposed code as written in 68000 assembly language. It is clear from this example that the proposed code is concerned only with passing parameters to and initiating startup of the coprocessor. All of the remaining functionality has been transferred. The three other entry points are similar in style.

```
unsigned _Send( rqst_msg, rply_msg,  id )
    /* Copyright National Research Council of Canada,  1983 */
    char        rqst_msg[], rply_msg[];
    unsigned  id;
  {
    extern struct TD *_Active;

    /* Set up td for receiver */
    _Disable();
    _Active->CORRESPONDENT = id;
    _Active->STATE = SENDING;
    _Active->REQUEST_MSG = rqst_msg;
    _Active->REPLY_MSG = rply_msg;

    _Block_signal_processor( id );

    _Enable();
    return( _Active->CORRESPONDENT );
  };
```

Figure 13. Original version of _Send() written in C

```
                    .text
                    .globl  _Send
*
*
      _Send:
            mtsr    #0x2600                 * disable
                                            * build _Comtab
            lea.l   _Comtab, a0             * ptr to _Comtab
            move.w  #CPSEND, (a0)+          * start code for cop
            move.l  (sp)+, (a0)+            * rqst ptr to _Comtab
            move.l  (sp)+, (a0)+            * rply ptr
            move.l  (sp)+, (a0)+            * id

            jsr     _Invoke_CP              * transfer to cop
*
            move.l  CORRESPONDENT(a0), d7   * return parameter
            jsr     _Enable
            rts

                    .globl  _Comtab
                    .globl  _Invoke_CP
                    .globl  _Enable
```

Figure 14. Proposed version of _Send() written in 68000 assembler

_Init_CP() is called during system startup and forces the coprocessor to transfer a number of constants to its global registers. These include pointers to _Active, to the first elements in the _Comtab, to the Mailbox, to the _Comdev (interrupt generator), to the _in_id_q, to the _Ready_q, and to other items as well. Having these pointers available within the global registers speeds up the operation of the coprocessor.

Microcode Configuration

Figure 11 shows the configuration of the proposed microcode. When idle, the coprocessor will execute _CPidle, which will be a tight polling loop which awaits a call from the 68000. When activated, _CPidle forces an arbitration cycle onto the 68000 which disables it. Next it fetches the start code from the _Comtab in main memory and branches to the appropriate microroutine: _CPsend, _CPreceive, _CPreply, _CPih, or _CPinit. Except for _CPinit, these microroutines are functionally identical to the primitives that originally executed on the main processor; only the speed of execution should differ.

The analysis undertaken thus far has led to the "pseudo-coding" of _CPidle, _CPsend, _CPblock_s, _CPipint, and a portion of _CPtd_serv. Other less significant routines have also been pseudo-coded in order to fully estimate system performance. The method of pseudo-coding is used here to mean the cycle-by-cycle description of the register transfer operations that are expected to occur on the actual hardware. Obviously this requires a sufficiently detailed description of the hardware to be meaningful. The process by necessity makes certain assumptions about the expected operation of the hardware and some of these may in fact be wrong. Nevertheless, the method is still very important in identifying problem areas in the design that must be improved as well as providing a good estimate of system performance.

A context switch is used for all transitions between microcode routines. As explained above, this provides not only a fast mechanism for communicating between routines and but also a clean set of registers with each call.

The typical flow of execution through a call to _CPsend will see a subsequent call to _CPblock_s which blocks the currently active task and generates an interprocessor interrupt. At this point two possibilities exist. First, flow may continue with a redispatch of the next task and a return back through _CPsend, _CPidle with the ultimate return of control back to the 68000. Second, an interprocessor interrupt may appear, possibly due to the fact that the preceding _CPsend was intended for a task that is resident on the same processor, in which case instead of returning directly

directly to _CPsend, a call occurs to _CPipint which services the interrupt. _CPtd_serv is called next to modify the appropriate task descriptors and flow returns through _ CPipint, _CPblock_s, _CPsend, _CPidle and eventually to the 68000. Calls to _CPreceive, _CPih, and _CPreply are expected to operate in a similar manner. It is worth re-emphasising that transitions between microcode routines are achieved in a single machine cycle or 100 ns.

The arrival of an interprocessor interrupt during the execution of the microcode is checked in each of the first level microroutines (_CPsend, etc.) as a last step before control is returned to the main processor. As shown in Fig. 11, this allows the immediate servicing of pending interprocessor interrupts without the expense of having to transfer back and forth between the coprocessor and the 68000.

5. A PERFORMANCE ANALYSIS OF THE COPROCESSOR

This chapter attempts to estimate the performance of the coprocessor by examining its expected behaviour at the register transfer level. The "pseudo-code" is concerned only with the operation of the execution unit, the performance of the control and interface units is transparent.

A Pseudo-code Example

A listing of _Send() as it appears in the current version of Harmony is shown in Fig. 13. Appendix A presents the corresponding pseudo-code. To simplify the code, main memory cycles are shown lumped, costing five machine cycles each. Pipelining is included in this example (pipelined operations appearing in the example are identified by a " ./. " symbol); however, the additional wait states introduced during over-the-bus memory cycles have been ignored in the analysis. The effects of delays that may be introduced because of forced memory refresh cycles on the OB68K1A have also been ignored in the analysis. Appendix B lists the preliminary allocation of the global registers. These are used for fast access to pointers to various data structures in the kernel. At this time, some of the pointers address the high order word of the item being pointed to while others include pointers to both the upper and lower words. Clearly this is subject to change.

An Analysis of the Operation of the Coprocessor

A thorough analysis of the expected performance of the coprocessor requires extensive coding at the register transfer level. This is extremely time consuming and consequently only a selected portion of program execution has been analyzed at this level of detail. Nevertheless it is felt that this limited study sheds enough light on the performance of the machine that a reasonable estimate of its behaviour can be determined.

The example presented in Appendix A outlining the code for _CPsend requires a total of 114 machine cycles which at 100 ns each amounts to a run time of 11.4 µs. (This represents only a portion of the processing performed during a complete send cycle.)

The analysis here is a direct end-to-end comparison of the execution of the two implementations of the _Send() function to the point where the next task is dispatched. The analysis then continues when the sending task is redispatched. The data presented in Chapter 2 (Fig. 2) indicate that the total execution time from the beginning of the _Send() function to the redispatch of the next task was 410 µs. Later, _Send() resumes for another 17 µs before completing its execution; a total of 427 µs is required overall. The analysis for the coprocessor includes the execution of the new version of _Send() and the inclusion of _Invoke_CP(). A summary of the estimated execution times is presented in Table 1. The execution time from the calling point in main() to _Send() until the redispatch of the next task, Child(), is 117.0 µs. Later, with the redispatch of main, _Send() runs for a further 32.5 µs, bringing the total time to 149.5 µs. This represents an improvement of only 2.9 over the 68000 implementation, a disappointing result.

Several observations can be made here. First, the benefit of servicing the interprocessor interrupts in microcode is apparent. _CPipint executes in 5.8 µs as compared to 58.0 µs for the 68000 implementation. The difference is due to the low cost of the context switch and to the fact that the queue being serviced is in local memory. It should be understood that this example includes a number of off-board bus cycles which are required in the servicing of the mailbox and the interrupt generator. Second, the last portion of _CPipint includes a search through the _Ready_q. The same number of iterations was used in each case: seven. Of the 15.6 µs spent in this part of the code, 13.3 µs was used in the search. Without doubt, the most important observations concern the expense associated with memory cycles to main memory.

Machine Cycle Statistics

Of the 149.5 µs required to execute the code, 64.2 µs (42.9%) will run on the 68000 while 85.3 µs (57.1%) will execute on the coprocessor. At this time the upper limit of performance can be estimated by reducing the execution time of the coprocessor to zero. In this case we find that the ultimate performance of the system is 427.0/64.2 = 6.7. This illustrates overhead associated with the transfer mechanism that has been developed and is due primarily to saving and restoring the state of the 68000 (39.8 µs).

Table 1. A comparison of the performance of the two implementations

Function	Execution time (μs) With coprocessor	Execution time (μs) Original configuration
_Send()	11.6	-
_Invoke_CP()	20.6	-
_CPidle	1.5	-
_CPsend	11.1	-
_CPblock_s	16.6	-
Arrival of interprocessor interrupt		
Total execution time in this segment	61.4	108.5
_CPipint	5.8	58.0
_CPtd_serv	25.5	153.5
_CPadd_ready	8.6	13.5
_CPtd_serv	0.1	8.0
Total time in this segment	40.0	233.0
_CPipint	15.6	68.5
Total time from start of _Send()	[117.0]	[410.0]
Analysis continues at the redispatch of the sending task		
_CPsend	0.3	-
_CPidle	0.2	-
_Invoke_CP()	19.2	-
_Send()	12.8	-
Total time in this segment	32.5	17.0
Total overall execution time	149.5	427.0

Table 2 summarizes the statistics of the machine cycles of the coprocessor. In this example, a total of 853 cycles are used in the code of which 509 (59.7%) are concerned with memory cycles to main memory. An additional 344 non-memory related cycles are used as well. In evaluating the design, two new factors can be introduced. The first concerns the effect of redesigning the kernel to reduce the frequency of memory cycles to main

Table 2. Coprocessor machine cycle statistics

Function	Original Harmony[2]			Proposed Harmony[3]		
	Total cycles	Memory access cycles	Other cycles	Transferrable memory access cycles	Fixed memory access cycles	Total
_CPidle	17	5	12	-	5	17
_CPsend	114	80	34	48 (6)	32	72
_CPblock_s	166	109	57	80 (13)	29	99
_CPinit	214	105	109	80 (15)	25	149
_CPtd_serv	256	150	106	150 (25)	-	131
_CPadd_ready	86	60	26	60 (8)	-	34
Total	853	509	344	418 (67)	91	502
				158 (31.5%)		

[1] Notes: "other cycles" are fixed and appear in both configurations; "transferrable memory access cycles" assumes 200 ns for a long word read and 100 ns for a long word write; the numbers in parentheses specify the number of coprocessor machine cycles required to replace the transferrable cycles.

[2] Estimate of coprocessor performance with the existing implementation of Harmony.

[3] With changes to Harmony that reduce access to main memory.

memory, and the second considers the impact of the use of a 32-bit processor in place of the 16-bit 68000.

Possible Mechanisms for Improvement

Restructuring the Kernal

With almost 60% of the coprocessor's activity tied up in the execution of memory cycles, it would seem that a redesign of the kernel that transferred many of its data structures to the coprocessor's private memory would be very effective in reducing this bottleneck. This is true because memory access to local memory is five to ten times faster than main memory access and because many of the data structures in the kernel are only of interest to the coprocessor in any case. This includes most of the contents of the task descriptors, the ready queues, and other variables such as _Active, _Victim_id, etc.

Unfortunately, while this may be attractive in the single processor case, this is complicated in our situation which involves multiprocessors. The principal problem here relates to the access to task descriptors on remote processors. In the current implementation of the Harmony kernel, task descriptors must be visible to all processors; this implies that they must reside in main memory. A possible alternative, the details of which are beyond the scope of this report, employs the concept of the "virtual task descriptor" (VTD) in which the coprocessor maintains the task descriptors locally but that virtual task descriptors be constructed in the coprocessor memory of the remote (correspondent) processor. Each VTD need only be a fraction of the size of the actual task descriptor containing only those elements that are essential for intertask communication (CORRESPONDENT, STATE, etc.). The message structure would have to be extended to provide a pointer to the virtual task descriptor and some cost would be associated with the construction of a VTD.

Nevertheless the benefits of such an approach can be estimated. Ignoring the additional cost of the "virtual task descriptor," then the analysis indicates (Table 2) that of the 509 machine cycles that are concerned with memory operations, 91 must remain in order to access the mailbox, the interprocessor interrupt generator, and the communications table, while the remaining 418 can be replaced by 65 machine cycles that need only access local RAM. (In the analysis used here, 10 read cycles to main memory are replaced by two LRAM read cycles while 10 write cycles to main memory are replaced by one LRAM cycle. The difference is due to the need to buffer data fetched from the LRAM after a read cycle.) The end result is that instead of requiring 853 cycles, the code would run in 502 cycles. The proportion of memory access related machine cycles is reduced from 58.4% to 31.5%. In this example, the 68000 would continue to require 64.2 µs (56.1%) but the coprocessor would now only need 50.2 µs (43.9%) to execute. The overall execution time would be 114.4 µs (vs 149.5 µs) which shows an improvement of 3.7 times that available with the 68000 implementation alone.

On this basis the redesign of the kernel can hardly be justified, particularly given that we have ignored all overhead costs and also that it would likely be a significant undertaking yielding only a marginal improvement.

The Impact of a 32-Bit Processor

Another consideration concerns the likely impact of the performance of a 32-bit processor, the 68020 for example. With a cycle time of 60 ns, a full 32-bit data path, and an architecture that includes an instruction cache, a conservative estimate suggests that a 68020 might execute the existing kernel at a rate of approximately twice that of the 10 MHz 68000.

(This should take into account wait states required because of memory cycle limitations.) It is probably reasonable to assume that the example discussed above will run in about 210 µs instead of 427 µs.

The impact on the hardware design of the coprocessor is limited by the fact that only the interface unit (IFU) and its microcode would have to be changed to accommodate a different processor. Changes to the memory data and memory address registers in the execution unit (EU) are minimal.

With a 32-bit interface, the coprocessor in its original configuration can be expected to require only half of the machine cycles that are needed in the design with the 16-bit 68000. Therefore, the machine cycle count associated with memory cycles is reduced from 509 to 255 and when added to the fixed cycles (344) amounts to 59.9 µs instead of the 85.3 µs seen previously. The anticipated performance based upon these assumptions shows that the combination of the 68020 with the coprocessor may execute the send cycle in 92.0 µs:

fixed cost: 64.2 µs/2 = 32.1 µs

total cost: 32.1 µs + 59.9 µs = 92.0 µs

instead of the 210 µs expected with the 68020 alone. This represents an improvement by a factor of approximately 2.4 over the 68020 alone. Table 3 summarizes the results.

Table 3. A performance summary

Source	Time (µs)	Speedup
68000	427	1.0
68000 With coprocessor	149.5	2.9
Coprocessor and changes to Harmony	114.4	3.7
68020	210 (approx)	2
68020 and coprocessor	92.0	2.4 (over 68020 alone)

The Impact of a Hardware FIFO

It is likely that the replacement of the message handling FIFO that is currently implemented in software with a hardware FIFO would result in some improvement in performance. At this time no effort has been undertaken to estimate the extent of that improvement since it too requires some redesign of the Harmony kernel.

6. CONCLUSIONS

From the results presented in Chapter 5 it appears that the level of performance that had been sought at the outset of the project cannot be achieved with the architecture described here. The principal limitations are related to the transfer mechanism between the processor and the coprocessor and the 16-bit data bus bottleneck that exists between the two processors. A proposal to minimize this bottleneck by re-configuring the kernel of the operating system is likely to offer only a slight improvement in performance.

The recent generation of 32-bit processors offer a number of features that will likely have an impact upon coprocessor performance. As an example, the 68020 features a coprocessor interface that permits concurrent operation of the two processors. In the case of the Harmony coprocessor, it could be argued that the overhead associated with saving and restoring the state of the 68000 in response to a call to the coprocessor could be reduced or eliminated entirely if the two processors operated in parallel. Unfortunately, with the current implementation of Harmony, this would likely produce a significant level of bus contention since both processors would be competing for bus cycles; the 68000 attempting to save its state while the coprocessor is fetching parameters and accessing the necessary task descriptors. The result could well be a net loss of performance.

In Chapter 5 it was noted that reducing the execution time of the coprocessor to zero would result in an upper limit of 6.7 in performance improvement for the 68000/coprocessor-based system. The limiting factor appears to be the cost of saving and restoring the state of the 68000. For the 68020, it is expected that saving the machine state will be at least as expensive as with the 68000 (on a machine cycle basis) given its more complex register structure. The use within the proposed execution unit of a RISC-like register file raises the question of the effects on performance of implementing the Harmony operating system on a RISC machine. RISC machines are particularly noted for fast context switching and therefore, all else being equal, would likely offer a major advantage in this application when compared to a general purpose processor like the 68000. While this issue remains to be explored it does offer an interesting alternative to hardware assistance.

The 68020 includes a 256 byte instruction cache that permits the simultaneous access of instructions from the cache and data from external RAM. As far as this report is concerned, the only impact of a cache is to modify the performance estimates given in Chapter 5 that assume a throughput improvement of about two of the 16.67 MHz 68020 over the performance of the

10 MHz 68000. The only meaningful way to determine the effect of a cache is to actually run the code and measure its performance. The behaviour of a processor with an instruction cache is related to the actual code being executed given the statistical nature of cache hits and misses.

Bus architectures are now appearing that provide enhanced access to local memory via memory expansion buses. Examples include the VMXbus (VMEbus), the MX-Bus (Matrox Electronic Systems Ltd.), and the iLBXII bus (Multibus-II, Intel Corporation). The analysis of the performance of a 68020-based system incorporating a coprocessor described in Chapter 5 assumed that one such high speed (zero wait state) 32-bit bus was available.

It had been hoped in this phase of the design that the development of a suitable architecture accompanied by only minor changes in Harmony itself would produce a significant improvement; this has not been the case. The design components that have clearly been helpful here include the use of a horizontally microcoded architecture that would allow the fully pipelined 100 ns cycle time, the use of overlapped memory mapped registers for fast context switching, and of course the 32-bit data path. This report has not attempted to address many of the engineering problems that would normally be associated with a project such as this. One such problem includes the development of a reliable interface between the 68000 processor and the coprocessor which must respect the electrical parameters that are critical to the proper operation of the 68000 (loading, delays, noise). Another concerns the development mechanism for microcode which includes either a writeable control store or a high speed ROM simulator with breakpoint and basic program trace features.

Ultimately the best solution is a combination of several concepts. First, a modified version of Harmony is required that utilizes virtual task descriptors that reside in the coprocessor's private memory. This not only results in fast memory cycles but also eliminates any possible contention with the main processor. Second, a more effective transfer mechanism is required to reduce main processor contention during calls to the coprocessor. One can imagine the use of a hardware FIFO as the basis of such a mechanism. Third, in order to eliminate the overhead of saving and restoring the state of the 68000, the two processors must be able to function concurrently. This is essentially a consequence of the first two concepts.

Despite the results obtained here, this exercise has been useful. It has clearly ruled out the initial concept and suggests that fundamental changes to the kernel of Harmony are necessary in order to achieve any significant increase in performance with hardware assistance.

REFERENCES

1. W.M. Gentleman. Using the Harmony operating system. NRC/ERB-966. Division of Electrical Engineering, National Research Council of Canada, Ottawa, Ont. December 1983, revised October 1984, May 1985.

2. S. Elgazzar, D. Green, and D. O'Hara. A vision-based robot system using a multiprocessing controller. NRC/ERB-971. Division of Electrical Engineering, National Research Council of Canada, Ottawa, Ont. June 1984.

3. D. Green. Chorus — a multiprocessor architecture for real-time control applications. NRC/ERB-964. Division of Electrical Engineering, National Research Council of Canada, Ottawa, Ont. December 1983, revised April 1984.

4. OB68K1A MC68000 single board computer user's manual. Omnibyte Corp., West Chicago, IL. 1983.

5. The 1240 logic analyzer operator's manual. Tektronix Inc., Beaverton, OR. 1983.

6. MC68000 16-Bit microprocessor. Motorola Semiconductor Products Inc., Austin, TX. April 1983.

7. Word Slice Product Family, Advance Product Information. Analog Devices, Digital Signal Processing Division, Norwood, MA. September 1984.

8. Am29332 Preliminary Specifications. AMD, Sunnyvale, CA. 1984.

9. D.A. Patterson and Carlo H. Sequin. A VLSI RISC. IEEE Comput. 15(9): 1982.

APPENDIX A. A LISTING OF "PSEUDO-CODE" FOR _CPsend

```
                PSEUDO-CODE     _CPsend                          CYCLE COUNT

*       comments are introduced by an asterisk
*
*       pipelined opertions are identified by:  ./.
*
*
*
* 1>    _Active->CORRESPONDENT = id;     ( source code )
*
*
        increment group generator - new context                      1
*
*       get "id" from _Comtab
*
        Y <- GR4                * GR4 has ptr to entry in _Comtab    1
        MAR <- Y                * pass to MAR                        1
        memory read cycle       * HI word
        ./. Y <- GR5
        ./. AR <- Y
        ./. IMDR <- data HI     * latch HI word
        ./. MAR <- AR           * adr of next word                   5


        memory read cycle       * LO word
        ./. IMDR <- data LO     * latch LO word                      5

        R31 <- IMDR             * store id in R31                    1

*
*       get pointer to active TD,  read contents of _Active
*
        Y <- GR6                                                     1

        AR <- Y                 * pointer to _Active                 1

        MAR <- AR               * adr of TD HI
        ./. I                   * "pre-immediate",  I = 2            1
        memory read cycle       * HI word
        ./. AR <- AR + I        * next address
```

```
        ./. IMDR <- data HI      * latch HI word
        ./. MAR <- AR            * load MAR                          5
        memory read cycle        * LO word
        ./. IMDR <- data LO      * latch LO word                     5
        R30 <- IMDR              * store ptr to TD in R30            1
*
*       determine address of _Active->CORRESPONDENT
*
        Y <- R30                 * ptr to active TD
        ./. I                    * offset to CORRESPONDENT           1
        AR <- Y + I              * form address
        ./. Y <- R31             * id to Y                           1
        MAR <- AR                * load address register
        ./. AR <- Y              * pass id to AR                     1
        MDR <- AR                * id to MDR                         1
        write cycle              * HI word                           5
        write cyle               * LO word
                                 * ( address manipulation assumed )  5
*
*
* 2>    _Active->STATE = SENDING;       ( source code )
*
*
        Y <- R30                 * adr of TD
        ./. I                    * offset to STATE                   1
        AR <- Y + I              * ptr to _Active->STATE             1
        MAR <- AR                * load MAR
        ./. I                    * value for SENDING                 1
        AR <- I                  * pass                              1
        MDR <- AR                * load MDR                          1

        write cycle              * HI word                           5
        write cycle              * LO word                           5
*
*
* 3>    _Active->REQUEST_MSG = rqst_msg;        ( source code )
*
*
*       get parameter "rqst_msg" from _Comtab
*
*
        Y <- GR0                 * ptr to _Comtab location containing
                                 * HI word of parameter "rqst_msg"   1
        MAR <- Y                 * transfer to MAR                   1
        read cycle               * HI word
```

```
        ./. Y <- GR1            * ptr to LO word of rqst_msg
        ./. MAR <- Y            * load MAR
        ./. IMDR <- data HI     * latch data                          5
        read cycle              * LO word
        ./. IMDR <- data LO     * latch data                          5
        R6 <- IMDR              * store rqst_msg                      1
*
*
*       get address of _Active->REQUEST_MSG
*
*
        Y <- R30                * adr of TD
        ./. I                   * offset to REQUEST_MSG               1
        AR <- Y + I             * ptr to _Active->REQUEST_MSG
        ./. Y <- R6             * value of rqst_msg                   1
        MAR <- AR               * load MAR
        ./. AR <- Y             * pass rqst_msg                       1
        MDR <- AR               * load MDR                            1
        write cycle             * HI word
                                * adjust adr for LO word              5
        write cycle             * LO word                             5
*
*
* 4>    _Active->REPLY_MSG = rply_msg;          ( source code )
*
*
*
*       get rply_msg from _Comtab
*
*
        Y <- GR2                * ptr to HI word of parameter
                                * in _Comtab                          1
        MAR <- Y                * load MAR                            1
        read cycle              * HI word
        ./. Y <- GR3            * ptr to LO word of param. in _Comtab
        ./. AR <- Y             * pass
        ./. MAR <- AR           * next adr
        ./. IMDR <- data HI     * latch data                          5
        read cycle              * LO word
        ./. IMDR <- data LO     * latch data                          5
        R6 <- IMDR              * store value of rply_msg in R6       1
*
*
*       form address of _Active->REPLY_MSG
```

```
*
*
        Y <- R30                * ptr to TD
        ./. I                   * offset to REPLY_MSG                   1
        AR <- Y + I             * adr of _Active->REPLY_MSG
        ./. Y <- R6             * value of rply_msg                     1
        MAR <- AR               * load MAR
        ./. AR <- Y             * pass rply_msg                         1
        MDR <- AR               * load MDR                              1
        write cycle             * HI word
                                * adjust address                        5
        write cycle             * LO word                               5
*
*
* 5>    _Block_signal_processor( id );          ( source code )
*
*
        * R31 contains id already
        * R30 contains pointer to TD already
        * a new context is established at the beginning of
          _Block_signal_processor()
        jsr     _Block_s        * as it is known                        1
        * on return from _Block_s the following IFU and CU
        * operations are to be performed:
          - check for a pending ipint and call _CPipint if necessary    1
          - decrement group generator to deallocate register set
            and branch to _CPidle at _CPret                             1
                                                                    =======
                                        Total cycles:                   114
```

APPENDIX B. PRELIMINARY ALLOCATION OF GLOBAL REGISTERS

```
GR0     pointer to 1ST parameter in _Comtab HI
GR1                    "                    LO
GR2     pointer to 2nd parameter in _Comtab HI
GR3                    "                    LO
GR4     pointer to 3rd parameter in _Comtab HI
GR5                    "                    LO
GR6     pointer to _Active
GR7     pointer to _Mailbox
GR8     pointer tn _Comdev ( interrupt generator )
GR9     my PE offset i.e. byte offset in _Comdev
GR10    base address of __In_id_q
GR11    offset to __In_id_p
GR12    offset to __In_id_c
GR13    pointer to __Ready_q
GR14    pointer to _Victim_id
GR15
 .
 .      not assigned
 .
GR31
```